MOVE GENERATION WITH PERFECT HASH FUNCTIONS

Trevor Fenner *Mark Levene*[1]

London, U.K.

ABSTRACT

We present two new perfect hashing schemes that can be used for efficient bitboard move generation for sliding pieces in chess-like board games without the need to use rotated bitboards. Moreover, we show that simple variations of these schemes give minimal perfect hashing schemes. The new method is applicable provided N, the number of k-bit spaced positions that may be set to 1, is not more than $k+1$. In chess, for a Rook's movement along a file $N = k = 8$; for a Bishop's movement $N \leq 8$, and $k = 9$ for a north-east diagonal and $k = 7$ for a north-west diagonal. The results of computational experiments comparing the efficiency of move generation with the standard method show that using the hashing scheme gives an average improvement of approximately 40%. The schemes we suggest are simple, efficient, and easy to understand and implement.

1. INTRODUCTION

A *bitboard* is a fixed-length sequence of bits used to represent a game board. In chess the length of a bitboard is conveniently 64 (8×8) (Slate and Atkin, 1977), in Shogi it is 81 (9×9) (Grimbergen, 2007), and in Gothic chess it is 80 (10×8) (Trice, 2004).

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	44	45	46	47
32	33	34	35	36	37	38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

Table 1: Chess bitboard notation.

A bitboard is represented as a vector of B bits, where bit 0 is the least significant bit. So, for example, in chess $B = 64$ and the bitboard can be presented in array format as shown in Table 1. (A *left shift* of the bitboard moves bit 1 to bit-position 0, bit 2 to bit-position 1, etc. We note that if the bitboard is considered to be the binary representation of the integer β, a left shift will divide β by 2.) Move generation using bitboards, for board games such as chess, can be implemented conveniently and efficiently by using logical bitwise operations for combining bitboards and masking the desired portion of the combined bitboard.

[1]School of Computer Science and Information Systems, Birkbeck College, University of London, London WC1E 7HX, U.K. Email: {trevor,mark}@dcs.bbk.ac.uk

(Chess has an advantage over, for example, Shogi and Gothic chess as its board size is ideal for efficient bitwise operations.) Consider the chess position from Reinfeld (1955) shown in Figure 1, with White to move. This chess position can be represented by several occupancy bitboards, one for each type and colour of piece, e.g., for the white Bishops just bits 17 and 38 will be set to one. We can then, for example, obtain a bitboard representing the occupancy of the pieces on the e-file by ORing all the occupancy bitboards and then ANDing the result with a bitboard representing the e-file, i.e., one with bits 4, 12, 20, 28, 36, 44, 52, and 60 set; this results in a bitboard with bits 4, 52, and 60 set. We refer to Cracraft (1984) for a detailed description of the bitboard technique, Heinz (1997) for an implementation of bitboards in computer chess, and Grimbergen (2007) and Trice (2004) for the use of bitboards in Shogi and Gothic chess, respectively. Bitboards have also been used to model the search space in combinatorial search problems such as the maximum clique problem (cf. San Segundo *et al.*, 2006).

To facilitate move generation in computer games such as chess, it is customary to precompute bitboards containing the possible movement of the various pieces from each of the 64 squares. The sliding pieces, i.e., Rook, Bishop and Queen, are more problematic than the other pieces since, in order to determine their possible movement, we need to know the configuration of pieces on the rank, file, or diagonal under consideration. So, for example, in Figure 1 the Bishop on b3 cannot move to c2, nor can the Bishop on c8 move to g4, while the Rook on e1 can move to e7 (taking the Knight) but not to e8. A common solution to this problem for sliding pieces is to precompute their movement from each square for all 256 possible occupancy configurations of the pieces along the rank, file, or diagonal. (Without loss of generality, we may ignore the fact that some diagonals are shorter and that, given the square of the moving piece, there are only 128 possible configurations for the other pieces.) Yet, there is a problem that still remains: given a bitboard representing all the pieces on a single rank, file or diagonal, how do we map this 64-bit vector to an 8-bit vector that determines which of the 256 occupancy configurations is present on the rank, file, or diagonal? The naive, but probably not the most efficient, solution is to loop over the positions in the rank, file, or diagonal.

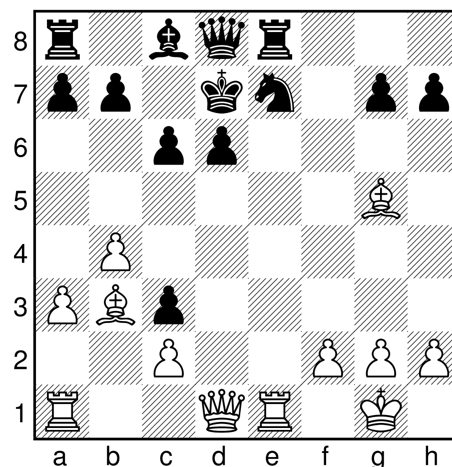

Figure 1: Example position illustrating rook and bishop movements.

Several solutions that avoid the looping have been proposed, the best known being the use of rotated bitboards, suggested by Hyatt (1999). A *file* bitboard corresponding to the *rank* bitboard in Table 1 essentially rotates the board by 90 degrees, so that bits are numbered consecutively along the files rather than the ranks; the two *diagonal* bitboards correspond to similar rotations by plus or minus 45 degrees. Rotated bitboards are efficient, but the maintenance of the additional bitboards gives rise to extra complexity and space requirements. An alternative to rotated bitboards is the method of magic move-bitboard generation (Kannan, 2007), which is based on ideas presented in Leiserson, Prokop, and Randall (1998) for creating perfect hash functions (Czech, Havas, and Majewski, 1997) in order to index those bits that are set to 1. We make use of the following two well-known concepts.

Definition 1: A *perfect hash function* is a one-to-one mapping from keys to addresses, i.e., different bit patterns are mapped to different addresses.
Definition 2: A perfect hash function is *minimal* if it maps the keys to a consecutive sequence of integers.

The hash functions used in magic move-bitboard generation are created by trial and error. Moreover, separate hash functions have to be generated for each square and piece; this is more complex than our method and their resulting hash tables may require more memory. A recent proposal (Tannous, 2007) is to use "direct lookup" to facilitate access to the ranks, files, and diagonals, instead of maintaining rotated

bitboards. Incremental maintenance of rotated bitboards is unnecessary under this scheme, which makes use of built-in associative arrays (provided by the programming language) that are implemented using a general hashing scheme. The perfect hashing schemes we present here, which efficiently map a 64-bit vector to an 8-bit one, provide a simpler, as efficient and more elegant solution, and like the above methods are also applicable to other board games. Moreover, we show that simple variations of these perfect hashing schemes are minimal.

The rest of the article is organised as follows. In Section 2 we describe two new perfect hashing schemes. In Section 3 we describe their application to move generation for sliding pieces in computer chess, present some experimental results, and compare our method to existing ones. In Section 4 we give our concluding remarks.

2. TWO PERFECT HASH FUNCTIONS

We use the symbol $\mathbf{mod}$ to denote the modulus function, i.e., $b \mathbf{\ mod\ } m$ is the remainder when using integer division to divide b by m. We employ the standard notation that for integers b, c and m, where $m > 0$,

$$b \equiv c \pmod{m}$$

means that b and c are *congruent modulo* m, i.e., $(b - c)$ is integrally divisible by m (Graham, Knuth, and Patachnik, 1994), or equivalently

$$b \mathbf{\ mod\ } m = c \mathbf{\ mod\ } m.$$

In this article, we consider a bitboard of which the least significant bit is bit-position 0 (as in Table 1), and where only N fixed positions that are k bits apart may be set to 1. These N positions are called *active* positions and the remaining positions, which are always set to 0, are called *inactive* positions. For simplicity, we assume that bit 0 is the first active position; so the set of active positions is

$$A_{kN} = \{0,\ k,\ 2k,\ \ldots,\ (N-1)k\}.$$

Without loss of generality, we could consider any bit-position to be the first active position by making a left shift that moves the first active position to bit 0. For example, if the active positions were all on the e-file rather than on the a-file (as in Table 2), then we could shift the bitboard 4 places to the left. This shift may be avoided, as explained at the end of this section.

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	44	45	46	47
32	33	34	35	36	37	38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

Table 2: Rook movement on the a-file.

For a chess bitboard we have $k = 8$ and $N = 8$ active positions for a Rook's movement along a file, as shown in bold in Table 2. Similarly, we have $k = 7$ and $N = 8$ active positions for a Bishop's movement along the main north-west diagonal, as shown in bold in Table 3 after shifting the bitboard 7 places to the

56	57	58	59	60	61	62	63
48	**49**	50	51	52	53	54	55
40	41	**42**	43	44	45	46	47
32	33	34	**35**	36	37	38	39
24	25	26	27	**28**	29	30	31
16	17	18	19	20	**21**	22	23
8	9	10	11	12	13	**14**	15
0	1	2	3	4	5	6	**7**

Table 3: Bishop movement on the *shifted* main north-west diagonal.

left. (Note that the shift results in bit 0 being set but not bit 56.) Correspondingly, for a north-east diagonal $k = 9$.

We construct two simple hashing schemes, one for $N \leq k$ (Subsection 2.1) and the other for $N \leq k + 1$ (Subsection 2.2). The first one is of particular interest when $N = k$ or $N = k - 1$, and the second when $N = k + 1$. Subsection 2.3 describes two minimal perfect hash functions.

2.1 The case $N \leq k$

We observe that

$$2^{nk} \equiv \left(2^k\right)^n \equiv (-2)^n \pmod{2^k + 2}, \tag{1}$$

since $2^k + 2 \equiv 0$ implies that $2^k \equiv -2$.

Now let $I \subseteq A_{kN}$ be the set of (active) positions that are set to 1, and let (a_n) be an indicator vector for I, i.e.,

$$a_n = \begin{cases} 1 & \text{if } nk \in I \\ 0 & \text{if } nk \notin I \end{cases}$$

We represent the bitboard corresponding to I by the integer α, where

$$\alpha = \sum_{i \in I} 2^i = \sum_{n=0}^{N-1} a_n 2^{nk}. \tag{2}$$

So there are 2^N possible values of α, since there are 2^N possible subsets I.

We now define the hash function h_1 by

$$h_1(\alpha) = \alpha \,\mathbf{mod}\, \left(2^k + 2\right), \tag{3}$$

so $0 \leq h_1(\alpha) \leq 2^k + 1$. Using (1), we then have

$$h_1(\alpha) \equiv \alpha \equiv \sum_{n=0}^{N-1} a_n 2^{nk} \equiv \sum_{n=0}^{N-1} a_n (-2)^n \equiv S(\alpha) \pmod{2^k + 2}, \tag{4}$$

where the function S is defined by

$$S(\alpha) = \sum_{n=0}^{N-1} a_n (-2)^n. \tag{5}$$

Therefore, by (3) and (4),

$$h_1(\alpha) = S(\alpha) \,\mathbf{mod}\, \left(2^k + 2\right). \tag{6}$$

We now show that h_1 is *perfect*, i.e., that it is a one-to-one function (Czech *et al.*, 1997). From (5), we have

$$S(\alpha) = \sum_{j=0}^{\lceil N/2 \rceil - 1} a_{2j} 2^{2j} - \sum_{j=0}^{\lfloor N/2 \rfloor - 1} a_{2j+1} 2^{2j+1}. \tag{7}$$

Now assume that $S(\alpha) = S(\beta)$, where

$$\beta = \sum_{n=0}^{N-1} b_n 2^{nk}.$$

It then follows from (7) that

$$\sum_{j=0}^{\lceil N/2 \rceil - 1} a_{2j} 2^{2j} + \sum_{j=0}^{\lfloor N/2 \rfloor - 1} b_{2j+1} 2^{2j+1} = \sum_{j=0}^{\lceil N/2 \rceil - 1} b_{2j} 2^{2j} + \sum_{j=0}^{\lfloor N/2 \rfloor - 1} a_{2j+1} 2^{2j+1}.$$

But, since integers have a unique binary representation, it must be the case that $a_n = b_n$ for all n, and thus the base -2 representation (5) is unique. It therefore follows that the function S is one-to-one.

By putting either $a_{2j} = 0$ and $a_{2j+1} = 1$, or $a_{2j} = 1$ and $a_{2j+1} = 0$, in (7) and summing the geometric series in each case, we obtain the following bounds on $S(\alpha)$:

$$-\frac{2}{3}(4^{\lfloor N/2 \rfloor} - 1) \leq S(\alpha) \leq \frac{1}{3}(4^{\lceil N/2 \rceil} - 1). \tag{8}$$

Whether N is even or odd, it is easy to verify that the there are precisely 2^N integers in this interval. Thus, provided $N \leq k$, the mapping $S(\alpha) \to S(\alpha) \,\mathbf{mod}\, \left(2^k + 2\right)$ will be one-to-one. So, since S is also one-to-one, it follows from (6) that h_1 will be one-to-one. Therefore, provided $N \leq k$, the range of h_1 will be 2^N integers in the interval $[0, 2^k + 1]$ excluding a single *gap* (subinterval) of length $2^k + 2 - 2^N$.

2.2 The case $N \leq k + 1$

The development follows along similar lines to that for h_1. We observe that

$$2^{nk} \equiv 2^{(k+1)(n-1)+k+1-n} \equiv \left(2^{k+1}\right)^{n-1} 2^{k+1-n} \equiv (-1)^k \, (-2)^{k+1-n} \quad \left(\mathrm{mod}\ 2^{k+1} + 1\right), \tag{9}$$

since $2^{k+1} + 1 \equiv 0$ implies that $2^{k+1} \equiv -1$.

We now define the hash function h_2 by

$$h_2(\alpha) = \alpha \,\mathbf{mod}\, \left(2^{k+1} + 1\right), \tag{10}$$

where α is as in (2), so $0 \leq h_2(\alpha) \leq 2^{k+1}$.

Using (9), corresponding to (4) we have

$$h_2(\alpha) \equiv \alpha \equiv \sum_{n=0}^{N-1} a_n 2^{nk} \equiv (-1)^k \sum_{n=0}^{N-1} a_n \, (-2)^{k+1-n} \equiv T(\alpha) \quad \left(\mathrm{mod}\; 2^{k+1}+1\right), \tag{11}$$

where the function T is defined by

$$T(\alpha) = (-1)^k \sum_{n=0}^{N-1} a_n \, (-2)^{k+1-n} = (-1)^N \, 2^{k+2-N} \sum_{m=0}^{N-1} a_{N-1-m} \, (-2)^m \, . \tag{12}$$

(The second summation is obtained by changing the index from n to $m = N-1-n$.) Therefore, by (10) and (11),

$$h_2(\alpha) = T(\alpha) \;\mathbf{mod}\; \left(2^{k+1}+1\right).$$

We now show that h_2 is a perfect hash function. As shown before, the base -2 representation (12) is unique, so T is one-to-one. If we define

$$T^*(\alpha) = (-1)^N \sum_{m=0}^{N-1} a_{N-1-m} \, (-2)^m \, ,$$

then $T(\alpha) = 2^{k+2-N} T^*(\alpha)$, so T^* is also one-to-one.

Corresponding to (8) we have

$$-\frac{2}{3}(4^{\lfloor N/2 \rfloor} - 1) \leq (-1)^N T^*(\alpha) \leq \frac{1}{3}(4^{\lceil N/2 \rceil} - 1). \tag{13}$$

Thus, provided $N \leq k+1$, the mapping $T^*(\alpha) \to T^*(\alpha) \;\mathbf{mod}\; \left(2^{k+1}+1\right)$ will be one-to-one. Since 2^{k+2-N} is co-prime to $2^{k+1}+1$ and

$$h_2(\alpha) = \left(2^{k+2-N} T^*(\alpha)\right) \;\mathbf{mod}\; \left(2^{k+1}+1\right),$$

it follows that h_2 will also be one-to-one provided $N \leq k+1$ (see Graham *et al.*, 1994, p. 125). Thus, provided $N \leq k+1$, the range of h_2 will be 2^N integers in the interval $[0, 2^{k+1}]$.

2.3 Two minimal perfect hash functions

Consider the hash function h_1. Since $N \leq k$, only 2^N of the $2^k + 2$ hash-table entries will be utilised. Apart from requiring significantly more memory than necessary when $N < k$, this may also affect the computation time if the larger table needs more cache lines. However, at the cost of an extra addition operation, we can significantly reduce the size of the hash table to exactly 2^N if we replace h_1 by the hash function $\widehat{h}_1$ defined by

$$\widehat{h}_1(\alpha) = \left(\alpha + \frac{2}{3}(4^{\lfloor N/2 \rfloor} - 1)\right) \;\mathbf{mod}\; \left(2^k + 2\right). \tag{14}$$

It follows from (4) and (8) that $\widehat{h}_1$ is a minimal perfect hash function with range $\{0, 1, \ldots, 2^N - 1\}$.

Now consider h_2. Since $N \leq k+1$, only 2^N of the $2^{k+1}+1$ hash-table entries will be utilised. Similarly, this requires significantly more memory when $N \leq k$, and may also affect the computation time.

Again, however, we can significantly reduce the size of the hash table to exactly 2^N at the cost of extra computation. Since $2^{N-1}T(\alpha) = 2^{k+1}T^*(\alpha)$, it follows from (11) that

$$(-1)^{N-1}2^{N-1}\alpha \equiv (-1)^{N-1}2^{N-1}T(\alpha) \equiv (-1)^{N-1}2^{k+1}T^*(\alpha) \equiv (-1)^N T^*(\alpha) \quad \left(\text{mod } 2^{k+1}+1\right). \tag{15}$$

If we now replace h_2 by the hash function $\widehat{h}_2$ defined by

$$\widehat{h}_2(\alpha) = \left((-1)^{N-1}2^{N-1}\alpha + \frac{2}{3}(4^{\lfloor N/2 \rfloor} - 1)\right) \ \mathbf{mod} \left(2^{k+1}+1\right),$$

then, using (13) and (15), we see that $\widehat{h}_2$ is a minimal perfect hash function with range $\{0, 1, \ldots, 2^N - 1\}$. The additional computational cost above that required to compute h_2 is an addition operation and a shift. However, the shift can be combined with the shift operation usually required to move the first active position to bit 0, in which case the computational cost of $\widehat{h}_1$ and $\widehat{h}_2$ will be approximately the same.

We now compare the hash functions we have defined in terms of their space and computational costs. When $N = k+1$, we would prefer h_2 over $\widehat{h}_2$, since the gap is only of length 1 and no extra addition operation is needed. When $N = k$, we would prefer h_1 over $\widehat{h}_1$ and $\widehat{h}_2$ for the same reasons, and over h_2 since this requires twice as much space for the hash table. When $N < k$, we would prefer h_1 over h_2 for the same reason; using h_1 does not entail the extra addition operation needed for $\widehat{h}_1$ and $\widehat{h}_2$, but at the cost of a much larger hash table. So if we wish to maintain a minimum size table in this case we would choose $\widehat{h}_1$, as $\widehat{h}_2$ sometimes requires an extra shift operation.

We note that we can relax the requirement that the first active position is at bit 0 if we are using h_2, and also for h_1 when $N \leq k-1$. This follows from the fact that in these cases the hash functions would still be perfect (see Graham *et al.*, 1994, p. 125). This would obviate the need to shift the first active position to bit 0.

3. APPLICATION TO MOVE GENERATION FOR SLIDING PIECES IN COMPUTER CHESS

In this section we consider a 64-bit bitboard representing a chessboard, as shown in Table 1. In this case $N = 8$, since this is the length of a rank, a file, and the maximum length of a diagonal. We note that, to map the bitboard representing the occupancy of the pieces on a *rank* to an 8-bit occupancy vector, all that is required is a single left shift that moves the first position on the rank to bit-position 0. The rank occupancy vector can then be used, together with the position of the square occupied by the Rook, to index a direct lookup table containing the possible moves of the Rook on the rank taking into account any other pieces present on the rank. (This table is normally pre-computed and then loaded into memory at startup to be used during move generation.)

3.1 k = N = 8 for the movement of a Rook along a file

This subsection corresponds to computing the possible moves of a Rook on a file taking into account any other pieces present on the file. For a Rook on a file other than the a-file, it is necessary first to left-shift the bitboard so that the first active position is at bit 0. For the a-file, the active positions are $A_{88} = \{0, 8, 16, 24, 32, 40, 48, 56\}$, as shown in Table 2, and the other positions of the bitboard are inactive. We use the hash function h_1 defined in (3), so the modulus is 258. It follows from (6) and (8) that there is a single gap of length 2 at the values 86 and 87. The hash function maps the bitboard representing the occupancy of the pieces on the a-file to a hash-table address. The hash-table entry at this address is an 8-bit occupancy vector representing the positions of the pieces on the file. This file occupancy vector can then be used, together with the position of the square occupied by the Rook, to index a direct lookup table containing the possible moves of the Rook on the file. Clearly, we could combine the hash table with the direct lookup table by using the hash address rather than the occupancy vector to index the lookup table. This would result in a further improvement in the efficiency of move generation.

3.2 k = N+1 = 9 for the movement of a Bishop along a north-east diagonal

This subsection corresponds to computing the possible moves of a Bishop along a north-east diagonal taking into account any other pieces present on the diagonal. For a Bishop on a diagonal other than the main north-east diagonal, it is necessary first to left-shift the bitboard so that the first active position is at bit 0. For the main north-east diagonal, the active positions are $A_{98} = \{0, 9, 18, 27, 36, 45, 54, 63\}$, and the other positions of the bitboard are inactive. We can use either the hash function h_1 or the hash function $\widehat{h}_1$, defined in (3) and (14), respectively, so the modulus is 514. (As explained above, the computational cost of $\widehat{h}_2$ will be similar to, but no better than, that of $\widehat{h}_1$.) It follows from (8) that for h_1 there is a single gap of length 258 from 86 to 343; for $\widehat{h}_1$ there is obviously no gap, since it is minimal. The possible moves of the Bishop may be found using the method described above for a Rook on a file.

3.3 k = N-1 = 7 for the movement of a Bishop along a north-west diagonal

This subsection corresponds to computing the possible moves of a Bishop along a north-west diagonal taking into account any other pieces present on the diagonal. It is necessary first to left-shift the bitboard so that the first active position is at bit 0. For the main north-west diagonal shifted 7 places to the left so that the first active position is at bit 0, the active positions are $A_{78} = \{0, 7, 14, 21, 28, 35, 42, 49\}$, as shown in Table 3, and the other positions of the bitboard are inactive. We use the hash function h_2 defined in (10), so the modulus is 257. It follows from (13) that there is a single unused address at 172. The possible moves of the Bishop may be found using the method described above.

3.4 Experiments

We now describe some computational experiments to compare the efficiency of move generation in chess with and without the use of these hashing schemes; when not using the hashing schemes we simply loop over the positions in the file or diagonal. The positions were taken from Reinfeld's *1001 Brilliants Ways to Checkmate* (1955). We tested the performance of pseudo-legal move generation on all of the 1001 initial positions, and repeated this 100 times to obtain consistent measurements. The computations were carried out in Matlab on a Windows XP platform, running on a desktop PC with an Intel Core 2 duo processor T5600, 1.83 Ghz, and 2 GBs of RAM. (Although a comparable implementation in a programming language like C would probably be faster than the Matlab implementation, the comparative performance should be fairly similar. We chose to use Matlab because of its convenience for experimentation.) We used the Matlab profiler to measure the performance; it produces detailed CPU measurements of all the functions called, allowing us to pinpoint the relevant lines of code and their respective timings. The results of our experiments, which are tabulated in Table 4, show an average improvement of about 40% when using the hash functions. We stress that the improvement is local to the routines that generate rook and bishop moves, and thus the global improvement to the pseudo-legal move generation will be significantly less.

Operation type	Number of calls	Without hash	With hash	Improvement
File (h_1)	237,600	7.801	3.775	51.61%
NE diagonal (h_1)	193,200	6.048	4.775	21.05%
NE diagonal ($\widehat{h}_1$)	193,200	6.048	5.097	15.72%
NW diagonal (h_2)	193,200	5.573	2.757	50.53%

Table 4: Total computation times in seconds and percentage improvement.

We note that the improvement of the operations along the north-east diagonals is less than half that of the others. This is mainly due to the fact that the computations were performed on a 32-bit processor and 64-bit division could only be done for signed integers. Since the north-east diagonal includes the sign bit, i.e.,

bit 63, this had to be explicitly catered for and involved additional computation. We also see that, when using the memory-efficient hash function $\widehat{h}_1$, the improvement is about 5% less than when using h_1, due to the extra addition operation. As mentioned in Section 3.1, combining the hash table with the direct lookup table would result in a further improvement.

3.5 Comparison with other methods

A similar improvement in efficiency (like that described in Subsection 3.4) over the standard method "without hash" would also apply to the rotated, magic bitboard, and direct lookup methods. As reported in Hyatt (2007), the rotated and magic bitboard methods are of comparable performance, and Tannous (2007) claims just a small improvement of the direct lookup method over rotated bitboards. It is easy to see that, in terms of the number of computer operations, the efficiency of our method will be similar to that of direct lookup. Thus we are justified in claiming that the computational efficiency of our method is comparable to the others.

It is true that there are open source programs, such as GPERF (Schmidt, 2000) and CMPH (Botelho, Pagh, and Ziviani, 2007), that will generate perfect hash functions for a set of keys. However, with GPERF minimality is not guaranteed, and the hash functions generated may not even be perfect. Alternatively, with CMPH the hash functions generated are more complex, much slower, and require linear space in the number of keys. Thus, the functions generated by these general purpose methods are less suitable for computer-chess engines due to their stringent efficiency requirements.

Our method improves on the state of the art by providing a general method for efficient bitboard move generation for sliding pieces (not restricted to chess) that: (i) avoids maintenance of auxiliary bitboards, as in the rotated bitboard method, (ii) does not depend on built-in system features, such as the associative arrays in the direct lookup method (and moreover these hash functions are not guaranteed to be perfect, so collisions may occur), and (iii) does not use a hash function generated by a non-trivial and possibly very long computation, such as the magic bitboard method, and which may not always be perfect for a given number of bits in the index (Kannan, 2007).

4. CONCLUDING REMARKS

We have presented two perfect hash functions (and variations of these that are minimal) that can be used to map the occupancy of pieces on a file or a diagonal from 64-bit bitboards to 8-bit occupancy vectors. The advantage of using our hash functions over existing methods is their simplicity, comprehensibility, ease of implementation, economy of memory usage, and the resulting efficiency of move generation, which is comparable to that of these methods. (Admittedly, our claims of simplicity and comprehensibility are necessarily somewhat subjective.)

We do not claim that our perfect hashing schemes will necessarily improve the performance of a computer chess program relative to the use of rotated bitboards (Hyatt, 1999), direct lookup (Tannous, 2007), or magic bitboards (Kannan, 2007). However, they are much simpler to implement than rotated bitboards as we do not need to store and maintain any auxiliary bitboards. Nor do they rely on system-provided hashing schemes as does the direct lookup method. Although not as general as the magic bitboards technique, we consider our hashing schemes to be simpler and more elegant, and they are often more space efficient; moreover, generating new hash functions requires no search or pre-computation.

We note that the results in Section 3 also apply to Rooks and Bishops in Shogi and Gothic chess, and to other games with sliding pieces on rectangular boards, provided the number of files is greater than or equal to the numbers of ranks. Our hashing schemes may also be applicable to non-rectangular boards with a regular grid structure, for example, hexagonal or triangular. Another possible domain of application is to binary feature selection from data sets represented by 0-1 matrices. To cater for the case when $N > k + 1$,

we could partition the bitboard into sufficiently small sections and apply one of our hashing schemes to each section; this would obviously involve additional computation. It is, however, an open problem to generalise our hashing schemes directly to bitboards that do not satisfy the condition $N \leq k + 1$.

5. REFERENCES

Botelho, F., Pagh, R., and Ziviani, N. (2007). Simple and space-efficient minimal perfect hash functions. *Proceeding of the International Workshop on Algorithms and Data Structures (WADS)*, pp. 139–150, Halifax, Canada.

Cracraft, S. (1984). Bitmap move generation in Chess. *ICCA Journal*, Vol. 7, No. 3, pp. 146–153.

Czech, Z. J., Havas, G., and Majewski, B. J. (1997). Perfect hashing. *Theoretical Computer Science*, Vol. 182, Nos. 1–2, pp. 1–143.

Graham, R., Knuth, D., and Patachnik, O. (1994). *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley, Reading, Ma., 2nd edition.

Grimbergen, R. (2007). Using bitboards for move generation in shogi. *ICGA Journal*, Vol. 30, No. 1, pp. 25–34.

Heinz, E. (1997). How DarkThought plays Chess. *ICCA Journal*, Vol. 20, No. 3, pp. 166–176.

Hyatt, R. (1999). Rotated bitmaps, a new twist on an old idea. *ICCA Journal*, Vol. 22, No. 4, pp. 213–222.

Hyatt, R. (2007). BitBoard Tests Magic v Non-Rotated 32 Bits v 64 Bits. `www.talkchess.com`. Posted 25 August.

Kannan, P. (2007). Magic move-bitboard generation in computer chess. See: `www.prism.gatech.edu/~gtg365v/Buzz/research/magic/Bitboards.pdf`. Last accessed July 2007.

Leiserson, C., Prokop, H., and Randall, K. (1998). Using de Bruijn sequences to index a 1 in a computer word. See: `http://supertech.csail.mit.edu/papers/debruijn.pdf`. Last accessed July 2007.

Reinfeld, F. (1955). *1001 Brilliant Ways to Checkmate.* Wilshire Books, Hollywood, CA.

San Segundo, P., Galan, R., Matía, F., Rodríguez-Losada, D., and Jiménez, A. (2006). Efficient Search Using Bitboard Models. *Proceedings of IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2006)*, pp. 132–138, Washington, D.C.

Schmidt, D. (2000). GPERF: A perfect hash function generator. *More C++ Gems* (ed. R. Martin), pp. 461–491. Cambridge University Press, Cambridge, U.K.

Slate, D. and Atkin, L. (1977). Chess 4.5: The Northwestern University Chess program. *Chess Skill in Man and Machine* (ed. P. Frey), pp. 82–118. Springer Verlag, NY.

Tannous, S. (2007). Avoiding rotated bitboards with direct lookup. *ICGA Journal*, Vol. 30, No. 2, pp. 85–91.

Trice, E. (2004). 80-square Chess. *ICGA Journal*, Vol. 27, No. 2, pp. 81–95.